CEWES MSRC/PET TR/98-37

Practical Aspects of Migrating DoD Codes to Scalable Architectures

by

S.W. Bova

H. A. Gabb

A. K. Stagg

07h00798

Work funded wholly or in part by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through

Programming Environment and Training (PET)

Supported by Contract Number: DAHC 94-96-C0002
Nichols Research

Practical Aspects of Migrating DoD Codes to Scalable Architectures

S.W. Bova* H.A. Gabb† A.K. Stagg‡

July 16, 1998

1 Introduction

At CEWES MSRC a tremendous amount of activity is currently underway to replace the aging vector supercomputers with the next generation, RISC-based scalable systems. This activity spans many levels and includes hardware acquisition, advanced training of the user base, and hiring of infrastructure personnel, application programmers, and numerical analysts. The CRAY-YMP was recently decommissioned, leaving the CRAY C90 as the only vector class machine at CEWES MSRC. This machine is also scheduled to be decommissioned in the near future. Simultaneously, millions of dollars have been spent acquiring scalable, RISC-based machines such as the IBM SP, the CRAY-T3E, and the SGI/CRAY Origin2000 systems. Future acquisitions of similar hardware are planned. The primary difficulty in using these new architectures is developing scalable software[1] which will run efficiently on multiple processors.

> Parallel algorithm design is not easily reduced to simple recipes. Rather, it requires the sort of integrative thought that is commonly referred to as "creativity." However, it *can* benefit from a methodical approach that maximizes the range of options considered, that provides mechanisms for evaluating alternatives, and that reduces the cost of backtracking from bad choices [1].

In this report, we discuss the issues associated with retrofitting serial or vector codes for the emerging class of scalable platforms at CEWES MSRC. Factors such as user demographics[2], adaptability of the algorithms to scalable platforms, and mission-critical requirements of the DoD should also be considered; however, we do not explicitly address these issues here.

*Mississippi State University On-site CFD Lead for PET, CEWES MSRC
†Computational Migration Group, CEWES MSRC
‡Computational Science and Engineering Group, CEWES MSRC
[1]Scalable software exhibits a resilience to increasing processor counts. A common example is the ability to increase the problem size at nearly the same rate as processors are added so that the time to solution remains approximately constant.

In general, we believe that there is some confusion within the DoD community regarding the processes involved in code migration. This paper is an attempt to clarify some of the pragmatic issues that arise when migrating vector code from the CRAY C90 to scalable, parallel systems. For example, we argue that simply providing services to parallelize user codes for distributed memory systems may be inappropriate. In general, the implementation costs required for algorithmic analysis and subsequent parallelization are much higher for distributed memory models than for shared memory models, and these costs must be carefully evaluated. Also, once a code has been parallelized, a user/developer is typically unable to make modifications and extend the code's life unless he has intimate knowledge of the underlying parallelization strategy. Instead, we advocate a cooperative relationship where code users/developers play an active role in the migration process to distributed memory systems.

2 Software Approaches to Parallelism

In this section, we discuss the most common models of parallelism. Our intent is not to give an exhaustive description of all possible approaches. The interested reader can see the text by Foster[1]. Instead, we provide a brief overview of the most popular techniques.

A myriad of software approaches are available to DoD users for parallelization of serial code. Factors such as the underlying algorithm and data structure, portability, performance and efficiency, maintainability, and implementation cost naturally will impact parallelization strategy. These strategies, which include MPI (Message-Passing Interface), HPF (High Performance Fortran), and OpenMP, may be broadly classified as either shared memory or message-passing.

> The essential feature of the message-passing model is that at least two processes are involved in every communication; send and receive operations must be paired. The essential feature of the shared memory model is that any process can access all the memory in the machine [3].

Another issue is whether process creation is static or dynamic. Figure 1 illustrates a code to be executed in parallel on four processors. Parallel implementations with tools such as OpenMP, PCF, and CRAY C90 multitasking directives are shared memory with dynamic process management. In this case, a single stream of execution (process 0) initially exists. This stream performs program initialization, *etc.*, until a `do-loop` associated with a parallelization directive is encountered. Then three more streams are created, usually by spawning additional processes. These "worker processes" share the address space of the master process and are destroyed when the parallel section of code is finished. If a second parallel section of code is encountered, new worker processes are spawned. In contrast, the situation is completely different for an MPI or HPF code. Four distinct execution streams exist from the start, and no address space

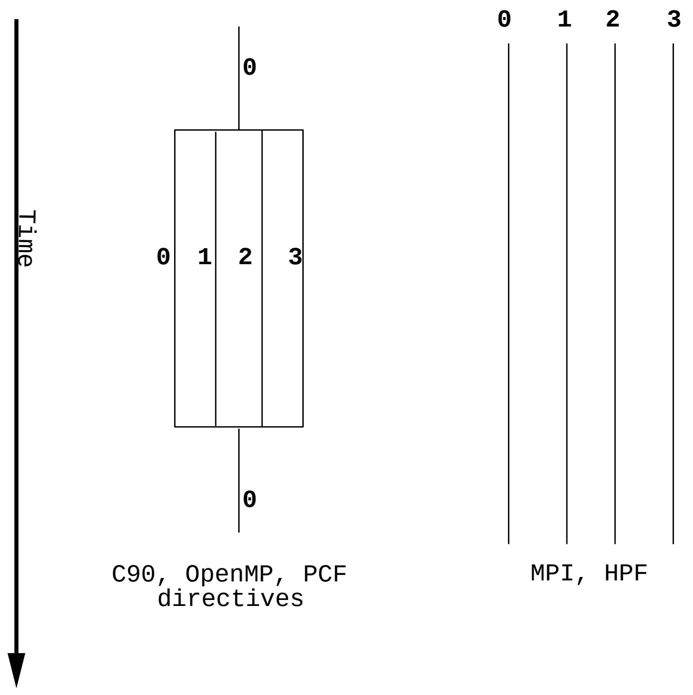

Figure 1: Static parallelism obtained via MPI and HPF versus multiple execution streams for loop-level dynamic parallelism obtained via compiler directives (CRAY C90, PCF, OpenMP).

is shared. The entire problem must therefore be manually distributed among the processors.

The illustration in Figure 1 is of course a generalization. Other approaches exist which may combine certain features. For example, PVM is a tool which allows a message-passing model with dynamic process management. The MPI 2.0 standard which was recently released also specifies dynamic process management, but vendor sources indicate that the first library releases are several years away. We emphasize that a distributed memory model can be used on a shared memory system. For example, this is the case when an MPI code is run on an SGI PowerChallenge. Also, in principle, a shared memory model can be used on a distributed memory system, although the software support for this is a more formidable challenge.

2.1 Shared memory

Implementing code for shared memory systems with compiler directives is relatively simple in principle. A global, shared memory makes it feasible for a migrator to insert compiler directives or C-preprocessor pragmas within the code to simply divide the work in loop structures across multiple processors. In fact, autotasking compilers used on machines like the CRAY C90 will automatically insert these directives within the code. Of course, one must take care to ensure that data dependencies within these loops are handled appropriately, or the parallelization may give erroneous results.

The nice feature of the shared memory model is that the parallelization process can be accomplished incrementally. A migrator can attack the code loop-by-loop without requiring any knowledge of the underlying algorithms. The migrator only needs to be aware of loop structures within the code and a few classic data dependencies [4]. Loops without compiler directives will simply be executed by a single processor. An advantage to this strategy when migrating autotasked CRAY C90 code is that the compiler directives can be mapped to other vendor compiler directives used on target scalable systems, as with PCF directives on the Origin2000. In fact, a preprocessor could be written using Perl or the Unix sed utility to automatically transform Cray compiler directives to PCF directives, for example. This would accelerate the migration of codes from the CRAY C90 to the shared memory environment of the Origin2000.

Posix threads (Pthreads) is another useful model for shared memory programming. Loop-level parallelism can be expressed using Pthreads, but the programming model is mainly task-parallel. Program functions are assigned to threads which are created by Pthreads library calls. Unlike compiler directives, all threads are peers. A thread may create new threads, terminate itself or other threads, signal other threads to terminate or begin working, etc. The operating system allocates resources to the threads, but the programmer must control synchronization and memory access. Compared to parallel compiler directives, Pthreads is low-level. However, a Fortran interface to the Pthreads library is not defined by the Posix standard. A Fortran application programmer interface to Pthreads is currently under development at CEWES MSRC [5].

The disadvantages of the compiler directive approach on shared memory systems are related to portability and performance. First, code modified with shared memory compiler directives will only run in parallel on systems whose compilers support that type of programming model. OpenMP is an attempt to alleviate this problem. However, the OpenMP specification does not yet define an interface to C or C++. Finally, parallel performance is lower in general with a shared memory model than with other approaches, partly due to the overhead associated with process creation and termination.

2.2 Distributed memory

In contrast with the shared memory programming model, implementing a code for the distributed memory environment with either explicit communication such as MPI or with HPF directives is more difficult. In the distributed memory programming paradigm, a code's data/memory must be explicitly distributed over the available processors by the programmer. This involves either tedious modification of the code for explicit communication when using MPI or careful insertion of data layout and parallelization directives when using HPF. Code parallelization cannot be accomplished incrementally in either case, and detailed knowledge of the code and its algorithms is necessary.

A general perception exists that using HPF to migrate a code is simpler than MPI and that the decreased implementation cost is worth the lower performance. This is not necessarily the case. For example, in the 1997 DoD Software Requirements Survey, 1,049 users claimed to require MPI, while only two listed HPF [6]. Although mitigating factors in the survey suggest that the discrepancy may not be this large, the numbers indicate that HPF is a viable option only in certain specialized cases where the generic data distribution patterns of HPF are applicable. For a discussion of the difficulties associated with certain types of data structures, the interested reader may consult reference [7].

HPF is similar to a shared memory model in that directives are used to describe data dependencies among processors, but with HPF the data must be distributed explicitly. For these reasons, parallelizing a code with HPF is not necessarily easier than developing an explicit message-passing version of the code, particularly for complex, irregular data structures. In fact, HPF may not even be a realistic option for codes with irregular data structures.

On the other hand, we feel that the implementation costs associated with message-passing are often greatly underestimated, even for codes with regular data structures. The problem is that a thorough analysis of the data flow and memory access patterns is required before the data structures can be partitioned and the communication calls added. The analysis must consider the entire algorithm, not just sections of code as would be possible if a shared memory model were being used.

One of the primary advantages of explicit communication models such as MPI is high performance with low communication costs relative to other programming models. Also, codes written in this way are generally more portable across both distributed and shared memory systems. In addition, explicit communication models can be used with Fortran, C, and C++ code since the communication libraries are developed separately from the compilers. The trade-off of course is in increased implementation cost.

3 Migration strategies

The optimal scenario is for original code developers to migrate their own codes with MSRC consultation. However, DoD users generally have constraints that limit their involvement in these activities. For example, if a DoD user is a government contractor, then he is generally employed under a specific contract whose scope does not include software optimization or porting. If the user is a government employee, he is usually in a production environment and does not have the luxury of optimizing or porting software. These difficulties are compounded by the steep learning curve associated with distributed memory computing.

Parallelizing software for distributed memory systems requires detailed knowledge of the underlying partitioning and communication strategies. Thus, it is vital that the DoD user be highly interested in learning the essential aspects of parallel algorithm design. For example, consider a code ported by MSRC personnel in the absence of such a commitment to knowledge transfer. When the DoD user later decides to upgrade the solver or the constitutive relations, additional communication calls or reevaluation of the original partitioning strategy might be required. If he is unaware of these issues, he likely will be unsuccessful in his modifications. He then has the option of learning the required material, returning the code to the MSRC for maintenance, hiring a consultant to perform the modifications, or reverting back to the original serial code and discarding the parallel code. These issues are especially acute if the code in question is a prototype with future modifications planned. In any case, if MSRC staff become involved in code parallelization without DoD user commitment to knowledge transfer, a dependency of the code developer upon the MSRC will be established automatically. The MSRC personnel involved in code migration must be aware of these issues and implications before attempting a port.

3.1 Selection

In consideration of the programming model issues discussed above, we propose a strategy for migrating serial code from the CRAY C90 at CEWES MSRC to scalable, parallel computers. Although other options are available, the main programming tools that we believe are realistic options are parallel compiler directives (PCF, OpenMP, and C-preprocessor pragmas), Pthreads, HPF, and MPI. PVM provides functionality similar to MPI, but vendor support is declining. Other approaches may be desirable on particular platforms. For example, SHMEM is a low-level Cray library which provides extremely high performance on the T3D and T3E family of systems by utilizing one-way communication between processors with a global address space.

We have attempted to identify the key issues involved in selecting a particular migration strategy. The underlying philosophy is an attempt to balance implementation cost against other issues such as performance and portability. (In this work we interpret the word *performance* in a narrow sense: we mean parallel scalability and floating point performance. We ignore issues such as message latency and bandwidth, *etc.*)

These trade-offs are illustrated qualitatively in Figure 2. Codes parallelized in shared memory with compiler directives such as PCF will incur the least implementation cost, be the least portable, and offer the lowest performance. If the OpenMP standard is embraced by most vendors, then we anticipate that its main benefit will be increased portability to other shared memory platforms but with no additional gain in performance. At the other extreme is MPI. This approach will provide the highest portability and good performance but at the highest implementation cost. Alternative approaches such as HPF and SHMEM also are available. With a low-level, vendor-specific library such as SHMEM, implementation cost is approximately as high as MPI, but portability is as low as PCF. The advantage of a vendor-specific tool like SHMEM is that in general the highest possible performance on a single architecture is achievable. HPF allows for moderate performance and portability which may be obtained

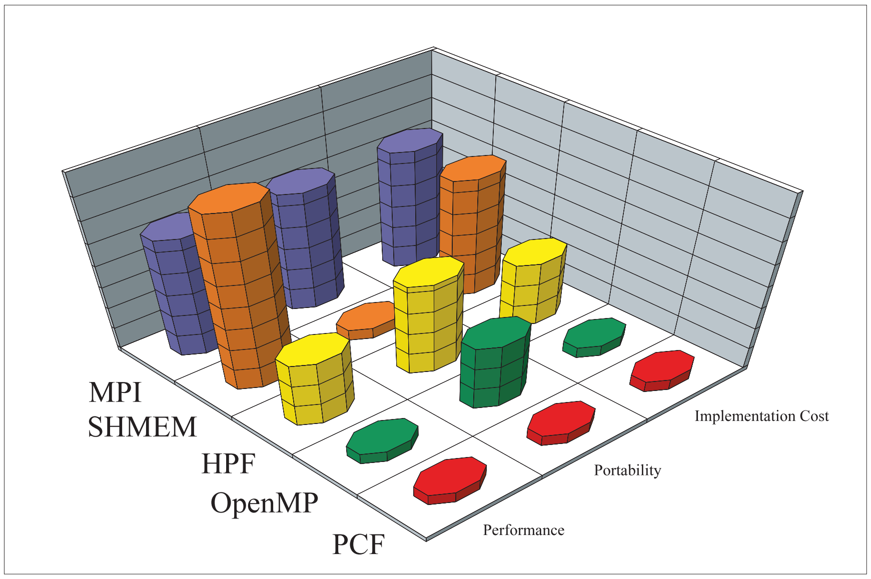

Figure 2: A qualitative illustration of the trade-offs among parallel performance, portability, and implementation cost for some popular parallelization tools.

at moderate implementation cost. In principle, HPF is more portable than OpenMP because HPF codes may run on both shared and distributed memory machines. We believe that HPF is less portable than MPI because in practice compilers implement only a subset of the HPF standard, and differences may occur from one architecture to another. Also, most commercial HPF compilers (including the ubiquitous Portland Group product) actually perform source to source translation and express parallelism using an underlying MPI or Pthreads library. For maximum performance and portability, we would advocate the use of MPI in all cases if implementation cost was not an issue.

In the remainder of this subsection, we propose a decision process which is illustrated in Figure 3. We emphasize that this is a general guideline only; unanticipated circumstances may suggest that an alternative approach be used. For example, if a user is working on a mission critical application that warrants extra attention, higher performance may be sought at any expense.

When migrating a code from the CRAY C90, a migrator should first determine if the application currently requires modern HPC resources. Computer hardware advances have resulted in enormous performance improvements in workstations, and in many cases these workstations will provide adequate resources. If this is the case, the application should be moved from the C90 to a suitable workstation. If the application actually requires HPC resources and is based on COTS (commercial, off-the-shelf) software, then placement of the code will depend on the migration activity of the 3rd-party software vendor. If the code does not depend on COTS software, then two choices exist. A functionally

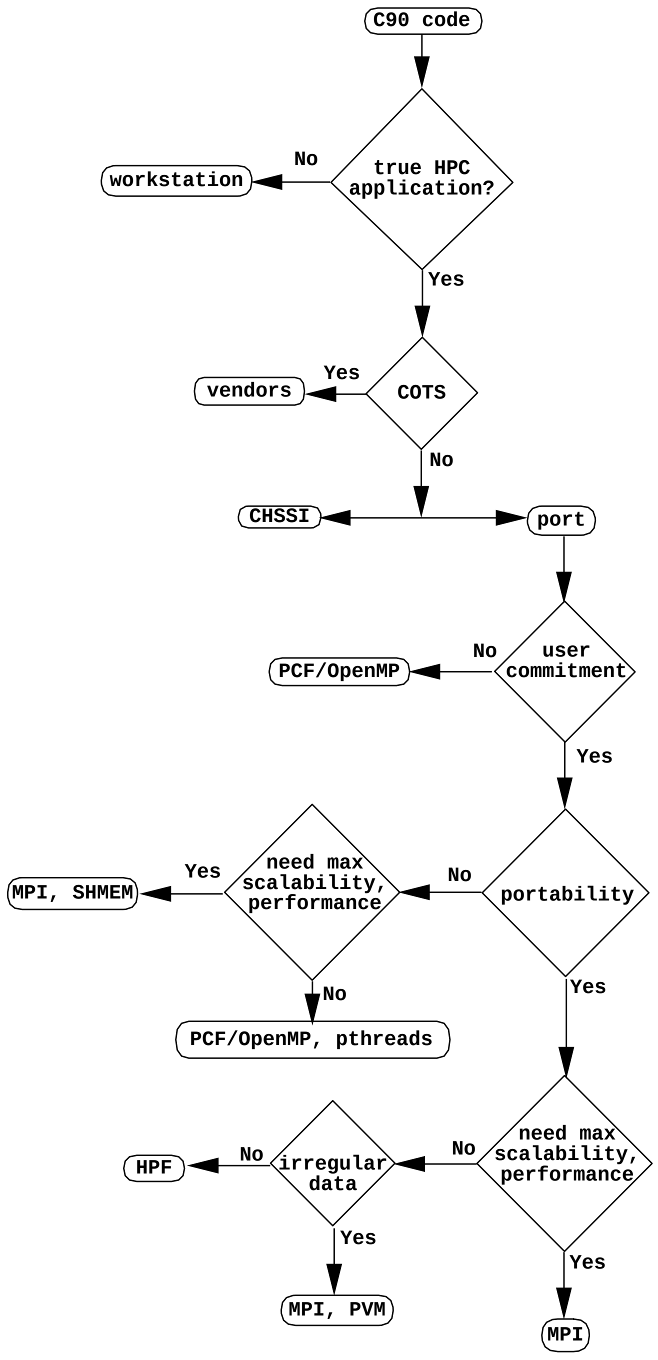

Figure 3: Proposed decision tree for migrating CRAY C90 codes.

equivalent CHSSI (Common High Performance Computing Software Support Initiative) code may be used if one exists, or the application can be ported to one or more CEWES parallel platforms.

The use of a CHSSI code is preferred for primarily two reasons. First, the CHSSI software development efforts are supported by the High Performance Computing Modernization Office (HPCMO) exactly for this purpose. Second, notwithstanding a user's preference to develop codes "in-house" (the "Not Invented Here Syndrome"), an existing portable, scalable, parallel code is easier to learn to use than it is to create a new parallel version from a serial vector code. Forty CHSSI codes span ten computational technology areas. Hence, for many DoD research projects, a suitable parallel application may already exist.

Without a suitable CHSSI code to provide equivalent functionality, serial code may be ported. When porting a code to CEWES parallel platforms, we feel that the user's level of commitment to involvement in the parallelization effort should play a key role in determining the model of parallelism used. When the user has not made a clear commitment to involvement in the parallelization effort, then we advocate that the code be parallelized using a shared memory model such as PCF or OpenMP primarily because of the relatively low implementation cost. If the user has committed to involvement, even if only to understand the parallelization work after it is accomplished, then the need for portability across CEWES MSRC systems should be assessed. If portability is unimportant but optimal performance is needed, then MPI or a low-level vendor library (such as SHMEM on the CRAY T3E) should be used. If neither portability nor performance is essential, then the use of a shared memory model such as PCF or OpenMP is a viable option. For cases where both portability and performance are most important, we advocate the use of MPI. On the other hand, if achieving maximum performance is relatively unimportant but portability is desired, then HPF is an option. However, HPF is only suitable for a relatively small subset of applications with regular data structures and memory access patterns. To achieve portability with codes having irregular, complex data structures, we recommend explicit message passing models such as MPI and PVM.

4 Conclusions

There is no precise recipe for parallelizing a serial code. Within the context of the CEWES MSRC, we have attempted to generalize the decision-making process. Before proceeding to parallelization strategies, it is necessary to verify that the problem still requires supercomputing capability. Then the two types of memory models, shared and distributed (Figure 1), are considered. Shared memory models are conceptually simple and more intuitive since all processes can access all data. This facilitates incremental parallelization. However, the cost of process creation/destruction along with the system overhead required to synchronize the memory access of parallel processes are a barrier to optimal performance. Distributed memory strategies give optimal performance, but often require low-level data manipulation. As a result incremental parallelization is difficult, and implementation costs are higher. A cost/benefit comparison of the major parallel programming models is given in Figure 2. A decision tree is given to help determine the best parallelization strategy (Figure 3). The decision tree attempts to reconcile the three main considerations: performance, portability, and cost of implementation. However, failure to consider the user's commitment to learn a particular parallel programming model can easily result in wasted effort on the part of CEWES MSRC personnel.

References

[1] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Publishing Company, Reading, MA, 1997.

[2] S.W. Bova, Wayne Mastin, and Carey Cox. A taxonomy of major CTA software at CEWES MSRC. Technical Report PET TR/97-01, CEWES MSRC, 1997.

[3] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, London, 1994.

[4] Kevin Dowd. *High Performance Computing.* O'Reilly and Associates Inc., Sebastopol, CA, 1993.

[5] Clay P. Breshears, Henry A. Gabb, and S.W. Bova. Towards a fortran 90 interface to the POSIX threads library. Technical Report PET TR/98-32, CEWES MSRC, 1998.

[6] Terry Clark. Analysis of DoD high-performance software development using the 1997 DoD requirements questionnaire. Technical Report DRAFT, ASC MSRC, 1997.

[7] S.W. Bova, Clay P. Breshears, and Henry A. Gabb. Status report on parallelization of MAGI. Technical Report DRAFT, CEWES MSRC, 1998.